A multidimensional array slicing DSL for Stream Programming

Pablo de Oliveira Castro, Stéphane Louise, Denis Barthou

A Multidimensional Array Slicing DSL for Stream Programming

Pablo de Oliveira Castro[1], Stéphane Louise[1] and Denis Barthou[2]

[1] CEA LIST, Embedded Real Time Systems Laboratory,
Point Courrier 94, Gif-sur-Yvette, F-91191 France
{pablo.de-oliveira-castro, stephane.louise}@cea.fr

[2] University of Bordeaux - Labri / INRIA
351, cours de la Libération, Talence, F-33405 France
denis.barthou@inria.fr

Abstract—Stream languages offer a simple multi-core programming model and achieve good performance. Yet expressing data rearrangement patterns (like a matrix block decomposition) in these languages is verbose and error prone.

In this paper, we propose a high-level programming language to elegantly describe n-dimensional data reorganization patterns. We show how to compile it to stream languages.

1 Introduction

Stream programming languages [1][2][3] are particularly well-suited to write efficient parallel programs for multi-core architectures. Fork-join parallelism and pipelines are explicitly described by the stream graph, and task memory requirements and communication costs may be statically extracted from the stream representation, enabling powerful optimization strategies [4][5] for high performance.

Languages such as StreamIt[1] or ΣC[2] are examples of stream languages with optimizing compilers. These compilers analyze the stream communication patterns and simplify them, breaking useless dependencies. These optimizations rely in particular on the fact that all communication patterns use Split, Duplicate and Join nodes. While very expressive, this low-level representation of dataflow reorganizations is very verbose and error prone.

In this paper we propose a high-level language for the description of stream reorganizations. In this language, streams are structured through iterators, enabling the construction of complex patterns of communication/reorganization. We show that these iterators and patterns can then be compiled efficiently into stream graphs using Split, Duplicate and Join nodes. The language can be seen as an extension to stream languages, as such we show how it can be integrated with the StreamIt language (but it could easily be adapted to other stream languages). We have implemented a compiler for this language that produces stream graphs.

1.1 Stream languages

Stream languages model parallel programs with stream graphs. In this dataflow representation, nodes represent either data reorganization operations between streams or filters, and arcs are communications between nodes. Each time a node is fired it will consume a fixed number of elements on its inputs and produce a fixed number of elements on its outputs.

Filters are particular nodes that have only one input and one output, they represent computation nodes, possibly keeping a state through successive firings. Split, Dup and Join nodes are nodes that dispatch data through the application. Since the focus of this paper is on data reorganization, we will concentrate on Split, Dup and Join nodes. We recall thereafter the main types of data reorganization nodes:

Join round-robin $\mathbf{J(c_1 \dots c_n)}$: a join round-robin has n inputs and one output. We associate to each input i a consumption rate $c_i \in \mathbb{N}^\star$. The node fires periodically. In its k^{th} firing the node takes c_u, where $u = (k \mod n)+1$, elements on its u^{th} input and writes them on its output. As in a classic Cyclo Static Data-Flow [6] model, nodes only fire when there are enough elements on their input.

Split round-robin $\mathbf{S(p_1 \dots p_m)}$: a split round-robin has m outputs and one input. We associate to each output j a production rate $p_j \in \mathbb{N}^\star$. In its k^{th} firing the node takes p_v, where $v = (k \mod m)+1$, elements on its input, and writes them to the v^{th} output.

Duplicate $\mathbf{D(m)}$ has one input and m outputs. Each time this node is fired, it takes one element on the input and writes it to every output, duplicating its input m times.

Besides, stream graphs have sources and sinks:

Source $\mathbf{I(l)}$: a source models a program input. It has an associated size l. The source node is fired only once and writes l elements to its single output. If all the elements in the source are the same, the source is *constant* and denoted by the node $\mathbf{C(l)}$.

Sink O: a sink models a program output, consuming all elements on its single input. If we never observe the consumed elements, we say the sink is *trash* and we write the node $\mathbf{T}$.

1.2 Motivating Example

As a motivating example we are going to present an excerpt from a matrix multiplication program that is shipped with StreamIt distribution 2.1.1. (cf. figure 1).

In StreamIt the stream graph is described hierarchically, in a textual form:

- *add*, is used to chain subgraphs.
- *split duplicate*, splits the previous output through a Duplicate node.

```
float->float pipeline
MatrixMultiply (int x0, int y0, int x1, int y1) {
 add RearrangeDuplicateBoth(x0, y0, x1, y1);
 add MultiplyAccParallel(x0, x0);
}
float->float splitjoin
RearrangeDuplicateBoth (int x0, int y0, int x1, int y1) {
 split roundrobin(x0 * y0, x1 * y1);
 // the first matrix just needs to get duplicated
 add DuplicateRows(x1, x0);

 // the second matrix needs to be transposed first
 // and then duplicated
 add RearrangeDuplicate(x0, y0, x1, y1);
 join roundrobin;
}
float->float pipeline
RearrangeDuplicate(int x0, int y0, int x1, int y1) {
 add Transpose(x1, y1);
 add DuplicateRows(y0, x1*y1);
}
float->float splitjoin
Transpose(int x, int y) {
 split roundrobin;
 for (int i = 0; i < x; i++) add Identity<float>();
 join roundrobin(y);
}
float->float pipeline
DuplicateRows(int times, int length) {
 split duplicate;
 for (int i = 0; i < times; i++) add Identity<float>();
 join roundrobin(length);
}
```

Fig. 1. StreamIt program for matrix multiplication

- *split roundrobin*, splits the previous output through a Split round robin node.
- *join roundrobin*, joins the previous outputs with a Join roundrobin node.

As we can observe in figure 1, describing reorganization of 2D data in StreamIt is quite fastidious.

2 HIGH-LEVEL LANGUAGE

We propose a high-level language that describes data reorganization operations on data streams, through the manipulation of shapes and slicing patterns. The language is build around five concepts: Shapes, Grids, Blocks, Iterators described in this section.

2.1 Shapes

The language restructures input streams into multidimensional patterns with *shapes* types. These shapes correspond to a multidimentional indexing of the stream elements.

In the following example, the two input streams, identified by the numbers 0 and 1 and accessed using the keyword "input", are structured into 3 shapes:

```
shape[10] A = input 0
shape[15,10] B = input 1
shape[3,3,3] C = input 0
```

Stream 0, is viewed in A as a stream of vectors of length 10, in C as a stream of $3 \times 3 \times 3$ cubes and stream 1 is viewed in B as a stream of 15×10 matrices.

More generally, given a view shape $[s_1, \ldots, s_d]$, the view coordinates $(x_1, \ldots, x_d)$ of the first pattern correspond to the linearized stream positions $\sum_{i=1}^{d} x_i * \prod_{j=1}^{i-1} s_j$. For later patterns, we must take into account the size of the previous patterns.

2.2 Grids

On instances of type $shape$ we can apply the grid operator which is defined by giving on each dimension i three parameters (l_i, h_i, δ_i):

- l_i is the lower bound of the grid for dimension i.
- h_i is the upper bound of the grid for dimension i.
- δ_i is the stride of the grid for dimension i.

For each dimension i, we consider the set of points :

$$G_i = \{\delta_i.k.\vec{e_i} : \forall k \in [|\frac{l_i}{\delta_i}; \frac{h_i}{\delta_i}|]\}$$

The elements of a grid are constructed by computing the Cartesian product of the G_i:

$$G = G_1 \otimes \cdots \otimes G_d$$

They are lexicographically ordered. This ordering defines a grid iterator $G(n)$, where $G(0)$ is the first element, $G(1)$ the second, etc.

The grid operator uses a standard slicing notation where l_i, h_i, δ_i are separated by colons and each dimension is separated by commas, $[l_1{:}h_1{:}\delta_1, \ldots, l_d{:}h_d{:}\delta_d]$.

The points described by the grid B [2:15:5,0:8:3] for instance are represented on figure 2(a). If the dimensions of a grid are not the same as the dimensions of the shape on which it is applied, a type error is raised. Out of simplicity, it is possible to omit one or more values of the triplet; missing values are replaced by sensible default values (0 in place of l_i, s_i in place of h_i, 1 in place of δ_i). For instance, the above example could be written B [2::5,:8:3] .

2.3 Blocks

The block operator can only be applied upon a grid type. A block is a d-dimensional box parametrized by its min and max coordinates on each dimension: $(a_1{:}b_1, \ldots, a_d{:}b_d)$ with $a_i, b_i \in \mathbb{Z}$.

$(-1:1, 0:1)$ defines a 3×2 block B, the points in B are lexicographically ordered, obtaining an ordered set:

$$B = \{(-1,0),(0,0),(1,0),(-1,1),(0,1),(1,1)\}_{lex}$$

Blocks must always be applied to a grid of same dimension using the product ($\times$) operator,

```
B[2::5,:8:3] x (-1:1,0:1)
```

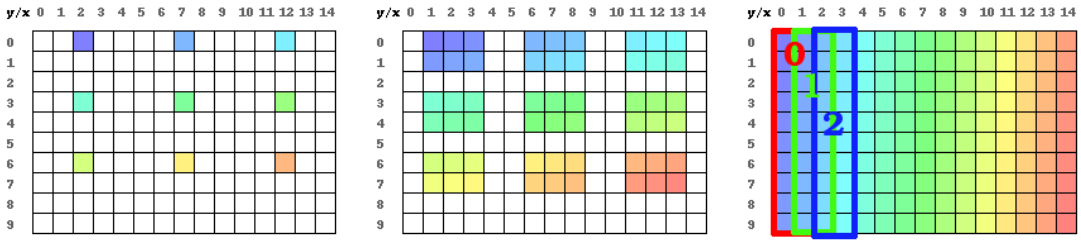
which describes the points in figure 2(b). If a block does not have the same dimension as the grid to which it is applied, a type error is raised.

To apply a block on a grid, we center the block around each point of the grid and take the resulting set of points. The resulting points, in order, are defined by the following iterator of ordered sets,

$$GB(n) = \{g + b : \forall b \in B\}_{lex}$$

Successive blocks may overlap, for example, B [::,0:1:] x (0:1,0:9) , extracts successive blocks of columns pairs from A (cf. fig. 2(c)).

When blocks fall partially or totally outside of the shape defined for the current stream, a configurable default value is returned for missing elements.



(a) B [2:15:5,0:8:3] (b) B [2::5,:8:3] x (−1:1, 0:1) (c) B [::,0:1:] x (0:1,0:9)

Fig. 2. Set of points described by, (a) a $grid[2]$, (b) a $gridblock[2]$, (c) an overlapping $gridblock[2]$. The gradient of colors gives the iterator order (cool colors are first).

2.4 Iterators

$Shape$, $grid$ and $gridblock$ are all instances of the $iterator$ type. We combine instances of the $iterator$ type to reorganize our data, using the "for", "in" and "push" keywords. The "for in" construct iterates over the elements of a given iterator. The "push" keyword produces an element or a ordered set of elements on the output.

```
shape[3] D = input 0
shape[2] E = input 1
 for d in D:
    for e in E:
     push e
     push d
```

produces the elements

$$\{E(0), D(0), E(1), D(0), E(0), D(1), E(1), D(1), \dots\}$$

2.5 Zipping

We introduce the zip polymorphic operator that enables us to interleave two iterators, or two ordered set of elements.

zip(A,B) interleaves the elements in the operands,

$$Z(n) = \begin{cases} A(\frac{n}{2}) & \text{if } n \equiv 0(mod.2) \\ B(\frac{n-1}{2}) & \text{if } n \equiv 1(mod.2) \end{cases}$$

2.6 Type system

The operators defined previously have a strict type system, ensuring that only correct programs are accepted:

$$\begin{aligned} grid, gridblock, shape &\in iterator \\ shape[s_1, \dots, s_d] &: shape[d] \\ [l_1{:}h_1{:}\delta_1, \dots, l_d{:}h_d{:}\delta_d] &: shape[d] \to grid[d] \\ (a_1 : b_1, \dots, a_d : b_d) &: grid[d] \to gridblock[d] \\ for.in &: iterator \to orderedset \end{aligned}$$

push and zip are polymorphic operators which we can both use on orderedset or iterators. push returns an IO type, since it pushes the elements in its operand to the output channel.

3 Compilation

This section presents the compilation of the high-level language introduced in the previous section into a stream graph. First we show that we can extract any $gridblock[1]$ using stream graphs. Then we compile $gridblock[d]$ graphs by composing multiple $gridblock[1]$ graphs. Finally we show how to handle "for in push" primitives.

3.1 Compilation of 1D gridblock

We observe that $[l : h : \delta]$ is equivalent to $[l : h : \delta] \times (0 : 0)$; therefore compiling $grid[1]$ instances is a special case of $gridblock[1]$ compilation.

We separate $gridblock[1] \equiv [l : h : \delta] \times (a : b)$ extraction in two steps:

- (cf. sec. 3.1.1), select the region $[l' : h' : 1]$ where $l' = l - a$ is the coordinate of the first element required, and $h' = l' + \delta.((h - l) \text{ div. } \delta) + b$ the coordinate of the last element.
- (cf. sec. 3.1.2 to 3.1.4), inside this region, extract the blocks $[:: \delta] \times (0 : w)$, where $w = b - a + 1$ is the width of the blocks.

Because a stream may contain an infinite sequence of patterns, it is important for the produced graphs to be reused an infinite number of times. We have ensured that after a pattern is consumed there are no left-over elements in any of the edges. This steady state execution guarantees that the graph can be reused without side-effects.

3.1.1 Selecting a region

We want to extract the region $[l' : h' :]$ from a $shape[1]$ of length s. If the region is $[0 : s :]$, we have nothing to do. In the other cases we must either cut some data (when the region is smaller than the shape) or inject some default values (when the region falls outside of the defined shape). These two cases can happen both for the upper or lower bound, we are going to detail the process for the lower bound:

- when $l' = 0$, we do nothing;
- when $l' < 0$ we inject $-l'$ default elements using a Join node and a constant Source;
- when $l' > 0$ we cut the first l' elements using a Split node and a trash Sink.

For $l' = -3$, $h' = 8$ and $s = 10$ we obtain the graph represented in figure 3.

Once the region is selected, a sequence of blocks can be extracted from it. Consider w, the width of the blocks,

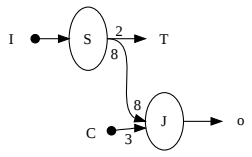


Fig. 3. Example of graph for grid region extraction

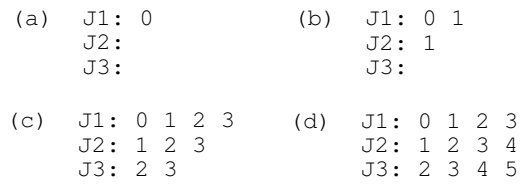


Fig. 5. Pipeline filling during complete overlap.

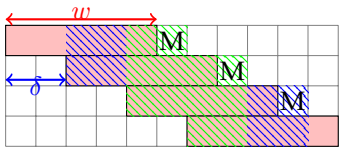


Fig. 6. Complete overlap with missing blocks (marked with an M)

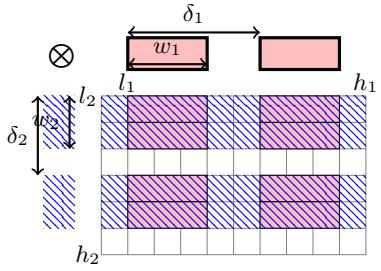


Fig. 7. Multidimensional region extraction

and δ, the stride of the grid. Depending on the ratio $\frac{w}{\delta}$ we can distinguish three situations:

- $\frac{w}{\delta} \leq 1$, no overlap, see section 3.1.2.
- $1 < \frac{w}{\delta} < 2$, partial overlap, see section 3.1.3.
- $2 \geq \frac{w}{\delta}$, complete overlap, see section 3.1.4.

3.1.2 *Extracting no-overlapping blocks (fig. 4(a))*

In this case, a sequence of n blocks of size w, separated by gaps of size $\delta - w$ must be produced in the stream. With a first split (S1) we extract the first block w, then $n-1$ blocks+gaps. The first block is produced on the output, then for each block+gap, we produce the block and throw away the gap with (S2+T).

3.1.3 *Extracting partial overlapping blocks (fig. 4(b))*

In this case, n blocks of size w, with overlaps of size $w - \delta$ (except for the first and last blocks) have to be extracted. We produce $w - \delta$ elements at the start of the stream (first edge of S1). After that we extract (second edge of S1) a sequence of $(n-1)$ chunks of δ elements (outlined in purple on the figure).

For each of these chunks, we separate (using S2) the overlapping (in stripped green) and non-overlapping parts. Both are produced, but the overlapping part is duplicated first (using D1). Finally we produce the remaining δ elements for the last block (using the third edge of S1).

3.1.4 *Extracting complete overlapping blocks (fig. 4(c))*

In the complete overlap case, we must produce n blocks, that are overlapped. This case corresponds to filling a pipeline. It is the most difficult of the patterns, because the number of nodes produced depends on the maximal number $m_{overlap}$ of overlapping elements. We show that $m_{overlap} = min(\lceil \frac{w}{\delta} \rceil, n)$. This overlap is reached once the pipeline is full (green stripped blocks), yet the pipeline must be filled and emptied (purple stripped and pink blocks). We are going to demonstrate how to achieve this when $m_{overlap} = 3$. The approach below can be generalized for any value $m_{overlap}$.

We start with $m_{overlap}$ joins (here J1, J2, J3). We start filling the pipeline, putting the first element in J1, as in fig. 5(a). Then we duplicate the second element twice (with D1) and put it in J1 and J2, as in fig. 5(b). The pipeline is now at full regime, each element of the stream is replicated three times (with D2) and put in J1, J2, and J3, as in fig. 5(c). Finally using D3 to duplicate 5, we fill the end of the pipeline, as in fig. 5(d). If we observe the pipeline matrix columns in fig. 5(d), we see that taking one element alternatively from rows J1, J2, and J3 produces the desired blocks on the output.

When $(w \text{ mod. } \delta) \neq 0$, we have missing blocks on the repetition pattern (cf. figure 6). To handle these missing blocks, we use a simple split and trash after the above pipeline pattern.

3.2 Compilation of Multidimensional Gridblock

Having shown how to generate any of the $gridblock[1]$, we now generalize the approach to higher dimensions.

Multidimensional grids and blocks, are by construction cartesian products of their 1D counterparts. For instance the $gridblock[2]$,

$$[l_1 : h_1 : \delta_1, l_2 : h_2 : \delta_2] \times (a_1 : b_1, a_2 : b_2)$$

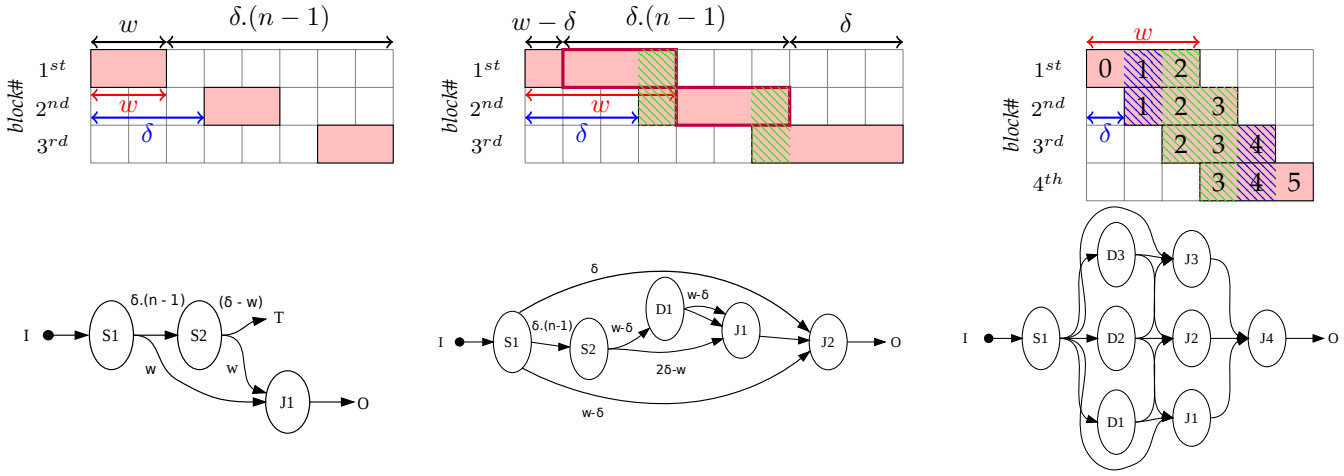
can be decomposed into,

$$([l_1 : h_1 : \delta_1] \times (a_1 : b_1)) \otimes ([l_2 : h_2 : \delta_2] \times (a_2 : b_2))$$

as shown in figure 7.

We are going to use this compositional property to compile $gridblock[d]$ graphs from a set of $gridblock[1]$ graphs:

1) We decompose the $gridblock[d]$ expression into its 1D components, $([l_i : h_i : \delta_i] \times (a_i : b_i))$, with $1 \leq i \leq d$.
2) We define the folded size f for dimension dim as:
$$f(dim) = \prod_{i=1}^{dim-1} s_i \quad \text{with } f(1) = 1$$
Which is the number of elements in any hyperplane obtained by cutting along the dim dimension.
3) We compile for every i, the graph G_i which produces the elements defined by,
$$[l_i.f(i) : h_i.f(i) : \delta_i.f(i)] \times (a_i.f(i) : b_i.f(i))$$



(a) No overlap ($\frac{w}{\delta} \leq 1$) (b) Partial overlap ($1 < \frac{w}{\delta} < 2$) (c) Complete overlap ($2 \geq \frac{w}{\delta}$)

Fig. 4. The three possible scenarios of $gridblock[1]$ extraction.

4) We chain the G_i graphs to produce the final graph,

$$G \equiv G_d \rightarrow G_{d-1} \cdots \rightarrow G_0$$

The obtained graph G extracts $gridblock[d]$. We will not prove it here for lack of space. The general idea is that each G_i extracts the 1D component for dimension i but is modified to consume elements of the folded size. Taking the example in figure 7, G_2 extracts $([l_2 : h_2 : \delta_2] \times (a_2 : b_2))$ (in the left margin of the figure), but considering elements of size s_1 (the row length). This process produces the striped region. Then G_1 extracts $([l_1 : h_1 : \delta_1] \times (a_1 : b_1))$ (in the top margin) with elements of size 1, effectively generating the expected blocks.

The above process extracts the elements using the lexicographic order of the shape. Nevertheless when working with $gridblock$, we must return the blocks on the order of the grid. To achieve this we add a reordering stage at the end of our graph (which is build with a Split followed by a Join).

3.3 Handling the "for in push" construct

Multiple streams can be combined by nesting different iterators. Our compiler analyzes the nesting level of each push and duplicates the elements on each stream accordingly, using a Duplicate and Join node.

Let us consider the following example:

```
for a in A:
  for b in B:
    for c in C:
      for d in D:
        push c
      push b
```

Let $\overline{X}$ denote the number of elements in iterator X. Given a *push x*, we define

- The iterator that generates x, is called the base iterator and noted $base(x)$. For instance, in the example above the base iterator of "push b" is B.
- $Outer(x)$ the set of iterators that contain $base(x)$. In the example above, $Outer(b) = \{A\}$ and $Outer(c) = \{A, B\}$.
- $Inner(x)$ the set of iterators that are contained by $base(x)$ and that enclose the statement "push x". In the example above $Inner(b) = \{C\}$ and $Inner(c) = \{D\}$.

We compute the length of $Outer(x)$, $\overline{Outer(x)} = \prod_{O \in Outer(x)} \overline{O}$ and similarly $\overline{Inner(x)} = \prod_{I \in Inner(x)} \overline{I}$. To satisfy the "for in" semantics we must replay the stream as many times as the number of iterations of the outer loops $\overline{Outer(x)}$; keep the current value steady for as many times as the number of iterations of the inner loops $\overline{Inner(x)}$. Both of these operations are easily expressed with a Duplicate node followed by a Join node.

3.4 Putting it all together

Combining region extraction and grid block extraction, for every program with a single push we can generate a stream graph. As multiple pushes can occur inside a loop, the stream graphs corresponding to the pushes are concatenated using a single Join node that gathers their outputs. Finally each zip is implemented using a single Split node that interleaves its inputs.

3.5 Optimizations

We have optimized our implementation to reduce the number of data copies and the number of nodes necessary for a pattern extraction.

To avoid unnecessary copies in the $griblock[1]$ graphs, the duplicate nodes have been placed at the later possible place, so that previous transformations are factorized (instead of duplicating a pattern and extracting a region twice, we extract the region once and duplicate it afterward).

To optimize $gridblock[d]$ graphs we choose an optimal order in which to chain the $griblock[1]$ stages. In this order all the no-overlap extraction stages (cf. section 3.1.2) are at the top of the pipeline. This is interesting because those stages throw away elements, therefore reducing the number of elements that have to be handled downstream.

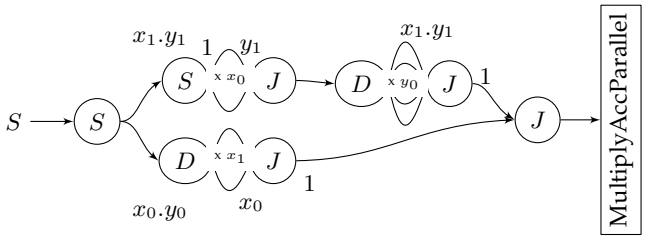


Fig. 8. Compiled graph for the matrix multiplication program in sec. 4

4 Matrix multiplication revisited

We define a new StreamIt keyword *datafilter*, which helps mixing our high-level language with normal StreamIt code. A datafilter, is like a filter, with the difference that it can have multiple inputs. It is instantiated with the join keyword *datafilter* as in below example:

```
float->float pipeline
MatrixMultiply (int x0, int y0, int x1, int y1) {
 join RearrangeDuplicateBoth(x0, y0, x1, y1);
 add MultiplyAccParallel(x0, x0);
}
(float, float)->float datafilter
RearrangeDuplicateBoth (int x0, int y0, int x1, int y1) {
  shape[x0,y0] A = input 0
  shape[x1,y1] B = input 1

  for l in A[0:1:,::] x (0:x0,0:0):
    for c in B[::,0:1:] x (0:0,0:y1):
      push zip(l,c)
}
```

The first input is seen as a stream of $x0 \times y0$ matrices, the second as a stream of $x1 \times y1$ matrices. A [0:1:,::] x (0:x0,0:0) iterates over the rows of A and B [::,0:1:] x (0:0,0: y1) over the columns of B. push zip(l,c) interleaves and yields (row, column) pairs.

The datafilter body is compiled using the method described the previous section, producing the graph in figure 8, which is in fact simpler that the graph that would have been generated by the original StreamIt program in figure 1 (the identity filters have been removed). StreamIt enforces that the stream nodes are connected hierarchically, but this it is not a requirement for the stream graphs produced in this paper.

We have not provided more examples for lack of space, but we have successfully used our proposed language to describe the data manipulations needed by a Sobel filter, a Gauss filter, a transposition, a FIR, etc.

5 Related Works

Dataflow programming models have been studied extensively [7], and many variants have been proposed. StreamIt[1] is both a stream language and an optimizing compiler for the RAW[4] and SMP[8] architectures. It is based on the cyclo-static dataflow model[6].

ZPL[9] is a parallel high-level language, allowing to globally describe the repartition of data on a set of processors and the communications between them. To describe data slicings, ZPL introduces a region first-class abstraction. Regions are n-dimensional arrays that can be moved around the original data using directions (stride vectors). The expressive power of ZPL in terms of region descriptions is comparable to our proposed language. Because there is less constraints on which directions are applied to a region, ZPL allows more complex walks through the data. Yet ZPL is not statically compilable to Stream graphs, indeed the order in which directions are applied depends on conditions which are not known until execution time.

Array-OL is a dataflow model and graphical programming language for signal processing [10]. The language is build around the concept of filters which are repeatedly applied. Each filter iteration space is defined with a pattern and a stride vector. The Array-OL model is similar to our model, but does not allow nested iterations over multiple streams.

The multidimensional shape type has been borrowed from the Single Assignment C (SAC) language [11], which proposes a functional C variant with multidimensional array operations. The main focus of this language is to optimize array manipulation by combining successive array operations into a single one.

Finally the slice notations used for grids and blocks are those used in the Matlab[12] and Python[13] languages.

6 Conclusion

In this paper we present a novel language for describing multidimensional data reorganizations. This frees the programmer from having to write complicated graphs with Join, Duplicate and Split nodes to express data reorganization patterns as they are described in our Domain Specific Language. We show how to compile this language to stream graphs and, as an example, integrate it with StreamIt. The graphs generated are optimized to avoid unnecessary data copies and are conservative in the number of nodes used.

References

[1] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proc. of the Intl. Conf. on Compiler Construction*, 2002.

[2] T. Goubier, F. Blanc, S. Louise, R. Sirdey, and V. David, "Définition du Langage de Programmation ΣC, RT CEA LIST DTSI/SARC/08-466/TG," Tech. Rep., 2008.

[3] C. Aussaguès, E. Ohayon, K. Brifault, and Q. Dinh, "Using Multi-Core Architectures to Execute High Performance-Oriented Real-Time Applications," To appear in *Proc. of Int. Conf. on Parallel Computing*, 2009.

[4] M. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[5] S.-w. Liao, Z. Du, G. Wu, and G.-Y. Lueh, "Data and Computation Transformations for Brook Streaming Applications on Multiprocessors," in *Proc. of the Intl. Symp. on Code Generation and Optimization*, 2006.

[6] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static Dataflow," in *IEEE Trans. on Signal Processing*, 1996.

[7] E. Lee and T. Parks, "Dataflow Process Networks," in *Proc. of the IEEE*, 1995.

[8] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe, "Cache Aware Optimization of Stream Programs," in *Proc. of the ACM conf. on Languages, Compilers, and Tools for Embedded Systems*, 2005.

[9] S. J. Deitz, B. L. Chamberlain, and L. Snyder, "Abstractions for dynamic data distribution," in *Proc. of the Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004.

[10] A. Amar, P. Boulet, and P. Dumont, "Projection of the Array-OL Specification Language onto the Kahn Process Network Computation Model," in *Intl. Symp. on Parallel Architectures, Algorithms and Networks*, 2005.

[11] S.-B. Scholz, "Single Assignment C: Efficient Support for High-Level Array Operations in a Functional Setting," in *J. Funct. Program.*, 2003.

[12] *MATLAB, Language Reference Manual v5*, The MathWorks, Inc., 1996.

[13] G. van Rossum, *Python Reference Manual*, CWI Report, 1995.